Grupa RSO3:
Damian Adamik
Tomasz Łukaszewski
Tomasz Mierzejewski
Piotr Mochocki
Konrad Rymuza
Witold Skomra
e-mail: T.Mierzejewski@elka.pw.edu.pl

Dokumentacja do projektu RSO

Spis treści

1. Protokół

1.1. Ustalenia ze spotkań i postępy

07.03.2006

Powstanie zespołu RSO3 i przydzielenie ról projektowych.

Kierownik zespołu – Tomasz Mierzejewski
Zarządzanie repozytorium i przechowywanie backupu – Witold Skomra
Przygotowanie demonstacji prototypu oraz prezentacji końcowej – Tomasz Łukaszewski
Zarządzanie dokumentacją – Piotr Mochocki
Testowanie rozwiązania – Damian Adamik

19.03.2006

Założenie repozytorium.

21.03.2006

Do zespołu dołącza Konrad Rymuza. Przejmuje zadania Damiana, który nie uczestniczy w pracach grupy.

28.03.2006

Zakończenie pierwszego etapu projektu.

11.04.2006

W kwestii MySQL-a i OpenSSI myśleliśmy nad systemem z kilkoma interfejsami do których odwołania kierowane byłyby na zmianę przez DNS lub podobny system instniejący w OpenSSI, partycjonowaniem takim jak udostępnia to klastrowy MySQL oraz potrójnymi replikami na każdą partycję.

Pytania i zagadnienia, na które musimy znaleźć odpowiedź zanim ruszymy dalej:

— możliwość użycia DNS-u do rozdzielania zapytań, postawienie demonów na głównych elementach klastra (3 maszyny)

— jak działa "heartbeatźaimplementowany w OpenSSI ?
— load-balancing między replikami
  – jaki poziom spójności ?
  – jak działa dynamiczna replikacja w MySQL ?

25.04.2006

Load-balancingu w MySQLu w zasadzie nie ma. Wszystko idzie przez główny węzeł danej grupy. Jeśli taki węzeł padnie to pozostałe wybierają spośród „żyjących” nowy węzeł główny. Jeśli chodzi o poziom spójności to przypuszczamy (dokumentacja milczy na ten temat) że jest on sekwencyjny.

29.04.2006

Koncepcja rozwiązania:

— maszyny obsługujące zapytania stawiamy na wydzielonych węzłach (głównych) OpenONE
— zapytania do nich kierujemy zgodnie z algorytmem karuzeli (round-robin) używając dynamicznego DNS-u
— tabele bazy danych znajdują się w partycjach
— każdą partycję obsługują trzy maszyny
— w ramach partycji dane są uspójniane i dopiero następuje powrót sterowania
— w razie padu którejś z maszyn partycja staje się read-only i teraz możliwe są:
  – oczekiwanie na dołączenie maszyny do grupy
  – działanie w mniejszej grupie i powrót do trybu read-write Oczywiście sytuacja taka powinna być zasygnalizowana operatorowi.

22.05.2006

Przydział zadań do trzeciego etapu projektu:

1. Rozpoznanie systemu loadbalancingu na OpenSSI - czy da się z niego skorzystać, jak to zrobić; jeśli nie, to zaproponowanie alternatywnego rozwiązania – Konrad Rymuza
2. Napisanie kodu do powyższego – Konrad Rymuza
3. Opracowanie protokołu replikacji dla silników bazy danych – Tomasz Łukaszewski
4. Zaimplementowanie daemonów sterujących silnikami bazy danych, relizujących powyższy protokół – Tomasz Łukaszewski
5. Opracowanie i wykonanie testów – Witold Skomra
6. Napisanie dokumentacji – Piotr Mochocki
7. Przeprowadzenie prezentacji gotowego rozwiązania – Tomasz Mierzejewski i Witold Skomra

2. Faza I

2.1. MySQL — paczka nr. 1 (T.M.)

2.1.1. Źródła

W niniejszym opracowaniu używam źródeł MySQL-a pobranych z serwisu *mysql.com* oznaczonych jako *mysql-5.1.7-beta.tar.gz*.

2.1.2. Kompilacja

Po rozpakowaniu paczki poleceniem

```
tar zxvpf mysql-5.1.7-beta.tar.gz
```

należy, w katalogu *mysql-5.1.7-beta*, dokonać konfiguracji wydając np. poniższe polecenie, które:

— określa położenie całości instalacji w podanym katalogu (opcja *--prefix*),
— buduje jedynie biblioteki ładowane dynamicznie pomijając statycznie,
— każe używać pliku gniazda o określonej nazwie (zamiast domyślnej */tmp/mysql.sock*).

```
./configure --prefix=/home/rso/tmierzej/mysql --enable-static=no \
    --with-unix-socket-path=/tmp/mysql.rso3.sock
```

Następnie należy wydać polecenie zbudowania MySQL-a. Wersja *5.1.7-beta* posiada błąd w skryptach konfiguracyjnych, który najłatwiej obejść stosując podany poniżej ciąg komend. Po poprawnym zbudowaniu, MySQL zostanie zainstalowany w wybranym podczas kofiguracji katalogu.

```
make
cd libmysql
make get_password.o
cd ..
make
make install
```

2.1.3. Konfiguracja rozruchowa

Poniżej przedstawione są polecenia służące do wstępnego skonfigurowania MySQL-a i zaczerpnięte są z dokumentacji tegoż pakietu.

```
cd mysql/
cp ./share/mysql/my-small.cnf ./var/my.cnf
./bin/mysql_install_db --user=tmierzej
./bin/mysqld_safe &
./bin/mysqladmin -u root password '<hasło administratora>'
```

2.1.4. Uruchomienie i zatrzymanie

Bazę danych uruchamia się wydając następujące polecenie

```
./bin/mysqld_safe &
```

a działającego MySQL-a można zatrzymać poprzez wpisanie

```
./bin/mysql_zap
```

2.2. Opis i analiza już istniejących rozwiązań w ramach MySQL umożliwiających realizację klastra (W.S.)

2.2.1. NDB

W rozwiązaniu klastrowym MySQL stosowanym storage engine jest NDB. Bazuje on na „*Pluggable Storage Engine Architecture*", czyli architekturze pozwalającej na łatwą wymianę silników baz danych, ze względu na jasne określenie interfejsu jaki musi być dostarczony przez implementacje. W tej architekturze proces msql zajmuje się przyjmowaniem zleceń od klienta i opracowywaniem planu ich wykonania jako szeregu operacji podstawowych, natomiast wykonanie elementów tego planu jest delegowane do konkretnego storage engine ustawionego dla tablicy na której wykonywana jest operacja. Podstawowe cechy NDB to:

— w trakcie działania dane są przechowywane w pamięci. Od wersji 5.1.X możliwe jest przechowywanie kolumn na których nie ma założonych indeksów na dysku
— kopiowanie danych na dysk jest asynchroniczne — trwałość danych wynika z założenia o istnieniu przynajmniej jednej repliki każdego elementu danych
— brak wsparcia dla konstrukcji foreign key. Jest to cecha wspólna wszystkich storage engine'ów stosowanych w MySQL
— możliwy jest cold start z dysku

Ndb wymaga aby każda przechowywana tablica posiadała klucz główny. W wypadku jego braku, tworzony jest sztuczny klucz ukryty. Wykorzystywana pamięć dzielona jest na dwa obszary: pamięć danych i pamięć na indeksy. Węzeł danych składa się z dwóch procesów: właściwego procesu ndb oraz tzw. anioła (*angel*) którego zadaniem jest kontrola stanu procesu ndb i w wypadku wykrycia jego zakończenia ponowne jego uruchomienie. To rozwiązanie wymusza ścisłą kolejność zabijania procesów. Proces ndb wykorzystuje tylko jeden wątek do operowania na danych oraz wykonywania innych czynności. Możliwość obsługi wielu żądań jest zrealizowana dzięki wykorzystaniu wywołań nieblokujących. Wątek wykonawczy nadzorowany jest przez watchdog'a w postaci drugiego wątku. Jego zadaniem jest

wykrywanie przypadków zablokowania się wątku wykonawczego w jednej z pętli. W wypadku wykrycia 3 (wartość ta jest zadana w kodzie ndb) „zatkań” (tj. braku lub zbyt późnym przekazaniu krążącego tokena) kończony jest cały proces. Ze względu na obecność „anioła” tak zakończony proces zostanie podniesiony i spróbuje ponownie dołączyć do klastra - ważne przy ręcznym „zabijaniu” procesów. Operacje skanowania wykonywane są równolegle we wszystkich fragmentach tablicy. W chwili obecnej ndb nie umożliwia przechowywania wierszy różnej długości. Oznacza to, że każde pole o zmiennej długości przechowywane jest jako pole o stałej długości równej maksymalnej długości danego pola. Przykładowo pole zadeklarowane jako VARCHAR(6) będzie przechowywane tak samo jak CHAR(6). Zbiór węzłów ndb odpowiada za przechowywanie danych. Dostęp do nich możliwy jest poprzez NDB API. Standardowo wykorzystuje się serwer msqld jako pośrednika, możliwa jest także bezpośrednia komunikacja użytkownika z procesami ndbd. Komunikacja między procesami ndb bazuje na wymianie komunikatów, nazywanych tutaj sygnałami. Może ona wykorzystywać jedno z poniższych mediów:

— TCP poprzez sieć ethernet. Minimalna wymagana przepustowość to 100Mbit/s. Ze względów bezpieczeństwa (dane przesyłane między węzłami nie są w żaden sposób chronione) i wydajności zalecane jest wykorzystanie dedykowanej sieci na potrzeby połączeń między węzłami. Aby zapobiec powstaniu tzw. „*single-point of failure*” zaleca się zdublowanie połączeń
— pamięć dzieloną w wypadku gdy procesy mają do takowej dostęp
— SCI

2.2.2. Architektura klastra

Z punktu widzenia aplikacji użytkownika, korzystanie z klastra nie różni się od korzystania z pojedynczego serwera MySQL. Jedyna różnica to możliwość wystąpienia dodatkowych nieudanych commitów, ale każda aplikacja korzystająca z DBMS powinna być przygotowana na konieczność powtórzenia transakcji. Podstawowym elementem klastra mysql jest klaster ndb odpowiedzialny za przechowywanie danych. W celu dostarczenia znanego interfejsu do komunikacji z klastrem wykorzystywany jest zbiór węzłów sql. W trakcie startu potrzebny jest też co najmniej jeden węzeł zarządzający, którego zadaniem jest udostępnianie konfiguracji startującym węzłom. Rozwiązanie klastrowe MySQL składa się z węzłów trzech rodzajów:

— danych (*data node*) - procesy ndbd, których powinno być nie mniej niż dwa. Są one odpowiedzialne za przechowywanie danych oraz za wykonywanie podstawowych operacji dostępu do nich (*scan, index scan*) w tym współpracę z innymi węzłami danych
— zarządzających - procesy *ndb_mgm* w klastrze musi występować minimum jeden węzeł tego typu w trakcie startu klastra. Teoretycznie nie jest potrzebny w trakcie działania, jednak w przypadku „padu” jakiegoś węzła i jego ponownego „wstania” będą mu potrzebne dane konfiguracyjne z węzła zarządzającego. Istnieje możliwość stosowania większej ilości węzłów tego typu, ale wymaga to zachowania pełnej zgodności konfiguracji wszystkich tych węzłów
— SQL; procesy *mysqld*, komunikujące się z procesami ndbd; ich zadaniem jest przyjmowanie zapytań, ich optymalizacja, rozbijanie na operacje pierwotne (scan, etc) i przesyłanie żądania ich wykonania do węzłów danych. W klastrze może występować dowolna liczba węzłów SQL

Na jednej maszynie może znajdować się dowolna ilość węzłów, choć ze względu na wydajność i niezawodność nie jest to zalecane. Wyjątkiem są węzły danych które ze względu na konstrukcję mogą zająć maksymalnie dwa procesory, więc w wypadku maszyn o większej ich liczbie wskazane jest uruchomienie kolejnych węzłów danych ale należących do innych grup. Węzły danych grupowane są w równoliczne (wymaganie mysql) grupy węzłów (*node group*). Grupa taka odpowiada za przechowy-

wanie replik jednej partycji. Grupa może zawierać od jednej (brak kopii) do czterech replik. Klaster pozostaje użyteczny dopóki w każdej grupie znajduje się przynajmniej jedna działająca replika. W wypadku „padu” wszystkich węzłów tracony jest dostęp do całej partycji, a w wypadku uszkodzenia dysków w tych węzłach dane z tej partycji są stracone. Nie jest obsługiwany *scheme discovery*, czyli operacje modyfikujące schemat, takie jak dodanie czy usunięcie tablicy, muszą być wykonane „ręcznie” na wszystkich węzłach danych/SQL wchodzących w skład klastra. Tablice dzielone są horyzontalnie według hasza z klucza głównego. Każda grupa węzłów odpowiedzialna jest za jeden fragment (partycje). Poszczególne węzły w grupie przechowują replikę fragmentu przydzielonego grupie. Podział na fragmenty jest automatyczny i przezroczysty dla użytkownika. Musi być zdefiniowany podczas konfiguracji bazy danych. Każdy z węzłów SQL łączy się z każdym z węzłów danych. W rezultacie operacja zlecana przez dowolny z węzłów SQL w klastrze jest wykonywana na tym samym (z dokładnością do awarii) zestawie węzłów danych, a więc po zakończeniu transakcji wszystkie węzły posiadają taki sam stan bazy danych. W przypadku awarii węzła podczas trwania transakcji jest ona wycofywana, a aplikacja kliencka jest o tym informowana. Każda aplikacja kliencka powinna być gotowa na powtórzenie transakcji. Wszystkie węzły muszą mieć przypisany w chwili startu numer ID unikalny w skali klastra. W klastrze ndb wykorzystywane są dwie metody detekcji awarii węzła: błąd połączenia i sygnał bicia serca.

Klaster ndb wykorzystuje protokół partycjonowania sieci, zapewniający, że w przypadku podziału klastra na części, na skutek awarii połączeń, tylko jedna część jest uznawana za aktywną i udostępniana do dostępu z zewnątrz. Węzły w pozostałych częściach są restartowane i dołączane do aktywnej części. W wypadku wykrycia błędu połączenia z węzłem wszystkie pozostałe węzły uważają go za niedziałający i ignorują go do momentu jego restartu. Stosowana jest zasada większości, to znaczy każda z części klastra ocenia czy stanowi większość oryginalnej konfiguracji. Stosowane są trzy kryteria:

— posiadanie więcej niż jednego węzła w każdej z grup
— posiadanie więcej niż połowy węzłów w grupach
— w wypadku remisu węzeł (zwyczajowo zarządzający) skonfigurowany jako mogący pełnić rolę arbitra wysyła własny głos

W celu wykrycia błędów takich jak problemy z dyskami, pamięcią czy też nadmiernym obciążeniem procesora, które sprawiają że węzeł przestaje poprawnie funkcjonować, ale nie naruszają połączenia wykorzystywany jest sygnał bicia serca. Węzły w klastrze ndb są zorganizowane logicznie w krąg, w którym każdy węzeł wysyła swój sygnał bicia serca do swoich sąsiadów. Okres co jaki jest on wysyłany, a także jakie są kryteria uznania węzła za niedziałający można ustawić niezależnie dla każdego węzła. Właściwe ustawienie tych wartości jest bardzo istotne gdyż przy nierozsądnym dobraniu tych parametrów można doprowadzić do sytuacji w której w pełni sprawny węzeł będzie po krótkiej chwili od startu klastra uznany za nieaktywny.

2.3. Wprowadzenie do systemu *subversion* (T.Ł.)

2.3.1. Wstęp

System kontroli wersji (ang. version control system) służy do zarządzania zbiorem plików wchodzących w skład projektu, czyli tzw. repozytorium. Główną różnicą pomiędzy systemem kontroli wersji a zwykłym serwerem plików jest wbudowany mechanizm śledzenia zmian w plikach i katalogach repo-

zytorium. System taki przechowuje **wszystkie** wersje plików, jakie kiedykolwiek zostały wprowadzone do repozytorium.

Pierwszy system kontroli wersji, *SCCS* (dołączony po raz pierwszy do wersji III systemu AT&T UNIX), przeznaczony był dla programistów, aby ułatwić śledzenie zmian w kodzie źródłowym. Od tamtej pory powstało wiele takich systemów, najbardziej popularne to *RCS* i *CVS* (systemy o otwartym kodzie) oraz *Rational ClearCase* (narzędzie komercyjne, obecnie własność firmy IBM).

System *subversion* był od początku projektowany jako narzędzie mające zastąpić system CVS. Autorzy postawili sobie za zadanie wyeliminowanie wad, jakie ma CVS, przy jednoczesnym zachowaniu maksymalnej zgodności na poziomie interfejsu użytkownika (głównie wiersza poleceń), aby ułatwić przejście użytkownikom przyzwyczajonym do systemu CVS. W świecie oprogramowania o otwartym kodzie źródłowym subversion jest obecnie drugim najczęściej używanym systemem kontroli wersji (po systemie CVS). Nowo powstające projekty najczęściej wybierają właśnie system subversion.

Celem niniejszego dokumentu jest przedstawienie systemu subversion z punktu widzenia końcowego użytkownika, który nie miał wcześniej do czynienia z systemami kontroli wersji, a chce (lub nawet musi) szybko zacząć z systemu subversion korzystać. Subversion jest narzędziem o bardzo dużych możliwościach – w tym opracowaniu przedstawione są jedynie podstawy, dokładny opis całego systemu znaleźć można w doskonałym podręczniku „Version Control with Subversion”, dostępnym pod adresem

`http://svnbook.red-bean.com/`

2.3.2. Architektura systemu

Najważniejszą, centralną częścią systemu subversion jest podsystem zarządzania repozytorium. Od wersji 1.1 systemu subversion zalecanym formatem przechowywania całego repozytorium jest baza danych *Berkeley DB*. Możliwe jest też utrzymywanie repozytorium w systemie plików serwera, ale ze względu na duże różnice w działaniu systemów plików na różnych platformach nie jest to zalecane.

Drugą istotną częścią systemu jest zestaw bibliotek klienckich. Odpowiadają one za komunikację z repozytorium (zarówno lokalnym, do którego jest bezpośredni dostęp, jak i zdalnym, z którym komunikuje się przez sieć TCP/IP), wymianę danych pomiędzy repozytorium a użytkownikiem i jego kopią roboczą. Interfejs użytkownika systemu subversion zbudowany jest w oparciu o ten zestaw bibliotek.

Jeżeli repozytorium jest udostępniane zdalnie, to potrzebny jest jeszcze trzeci komponent – serwer sieciowy. Dostępne są dwa rozwiązania – będący częścią systemu subversion serwer *svnserve* oraz moduł *mod_dav_svn* dla serwera *Apache*. Pierwszy z nich komunikuje się z bibliotekami klienckimi przy użyciu własnego, dedykowanego protokołu; drugi opiera się na protokole *WebDAV*.

2.3.3. Korzystanie z systemu subversion

Na potrzeby tego projektu umówione zostanie jedynie używanie systemu subversion z punktu widzenia użytkownika. Zarządzanie repozytorium, czyli wykonywanie wszelkich czynności administracyjnych, zostało wyznaczone jako jedna z ról w projekcie – pozostali członowie zespołu nie muszą być świadomi szczegółów tego zadania.

Repozytorium naszego zespołu dostępne jest lokalnie na klastrze, na którym pracujemy. Identyfikator określający miejsce przechowywania tego repozytorium to `file:///var/svn/rso3/SVN/rso3`.

Pierwsza kopia robocza

Pierwszą czynnością, jaką musi wykonać użytkownik, jest utworzenie lokalnej (we własnym katalogu) kopii roboczej:

`svn checkout file:///var/svn/rso3/SVN/rso3/trunk` *katalog*

W efekcie wykonania tego polecenia na terminalu powinniśmy zobaczyć listę pobieranych z repozytorium plików (lub odpowiedni komunikat o błędzie). Program *svn* jest podstawową, dostarczaną razem z całym systemem subversion, implementacją interfejsu użytkownika, obsługiwaną z poziomu wiersza poleceń powłoki. Słowo *„checkout”* jest poleceniem do wykonania – jednym z wielu dostępnych — i służy właśnie do tworzenia kopii roboczej wybranej *gałęzi* (ang. *branch*) repozytorium. Polecenie *checkout* jako swój argument przyjmuje identyfikator tej gałęzi – w naszym przypadku jest to identyfikator całego repozytorium z doklejonym na końcu słowem *„trunk”*, oznaczającym główną gałąź.

Wprowadzanie zmian

Ponieważ użytkownik pracuje wyłącznie na swojej kopii roboczej, wszelkie dokonane zmiany muszą być jawnie wprowadzane do repozytorium. W momencie, kiedy użytkownik uznajemy, że dokonane nas zmiany są gotowe, aby trafiły do repozytorium (czyli na przykład w momencie, kiedy kod daje się już skompilować), wykonujemy w katalogu roboczym polecenie

`svn commit` *nazwa_pliku*

W jego efekcie zmiany wprowadzone do pliku o podanej nazwie (lub wszystkie zmiany w kopii roboczej, jeśli nie podamy nazwy żadnego pliku) zostaną wprowadzone do repozytorium. Konieczne będzie jeszcze wprowadzenie komentarza opisującego dokonane zmiany. Komentarz nie powinien być zbyt szczegółowy, ale musi z niego jasno wynikać na czym polegała zmiana. Dobre komentarze bardzo pomagają w późniejszym przeszukiwaniu historii zmian dokonanych w projekcie, złe (nieużyteczne) komentarze powodują, że cały system kontroli wersji zamiast narzędziem wspomagającym staje się przeszkodą – **nie można** żałować czasu poświęcanego na odpowiednie opisywanie wprowadzanych zmian.

Odczytywanie stanu repozytorium

System kontroli wersji przeznaczony jest dla wielu użytkowników – członków zespołu projektowego. Każdy z nich wprowadza własne zmiany do repozytorium niezależnie od pozostałych, musi więc istnieć mechanizm synchronizowania zawartości własnej kopii roboczej z zawartością repozytorium. Do tego celu służy polecenie

`svn update` *nazwa_pliku*

Porównuje ono zawartość repozytorium i kopii roboczej i wprowadza w tej ostatniej zmiany dokonane przez innych użytkowników od czasu poprzedniego wykonania polecenia *update* (lub od momentu utworzenia kopii roboczej). Podając jawnie nazwy plików możemy ograniczyć działanie tego polecenia tylko do wybranej części kopii roboczej.

Sprawdzanie stanu kopii roboczej

Wydając w katalogu roboczym polecenie

`svn diff` *nazwa_pliku*

otrzymamy zestawienie wszystkich różnic pomiędzy aktualną zawartością określonego pliku w katalogu roboczym a jego najbardziej aktualną wersją znajdującą się w repozytorium.

2.4. Co trzeba wiedzieć o OpenSSI ? (K.R)

2.4.1. Wstęp

Ta część dokumentacji pomyślana jest jako spis zagadnień, których powinniśmy być świadomi w kontekście naszego projektu, w odniesieniu do szczegółowej dokumentacji OpenSSI przygotowanej przez naszych kolegów podczas poprzedniej edycji przedmiotu.

Z powodów praktycznych (zachowania czytelności tekstu) nie wszystkie cytaty zostały wyróżnione. Ewentualne nieoznaczone cytaty pochodzą z opisywanych w danym punkcie opracowań. Słowa mogą być przeredagowane, skrócone, tak aby w ścisłej formie przedstawiały istotę rzeczy.

2.4.2. Co to jest klaster ?

Opis wprowadzający do zagadnień związanych z klastrami zbudowanymi na bazie linuksa znajdziemy w pracach Anny Felkner [1] oraz Ewy Chachulskiej [2] pod wspólnym tytułem „Porównanie własności różnych dostępnych rozwiązań klastrowych zbudowanych w oparciu o system Linux”. Ze wstępu A. Felkner [1]:

Klaster to grupa niezależnych komputerów (wezłów), połączonych szybką siecią komunikacyjną. Z punktu widzenia użytkownika sprawia wrażenie pojedynczego systemu (komputera). Klastry pracują pod nadzorem specjalnego oprogramowania, które potrafi rozdzielać zadania między wszystkie pracujace w nim maszyny. [. . .]

Jednolity obraz systemu (ang. *SSI - Single System Image*) jest to fizyczny lub logiczny mechanizm dający złudzenie, że zestaw rozproszonych zasobów (pamiec, dysk, procesor) tworzy jednolity zasób. W obecnej chwili termin SSI nie ogranicza sie do jednego zasobu, ale jest coraz bardziej rozszerzany do wszystkich zasobów klastra. [. . .]

SSI może być wprowadzony w życie na kilku poziomach: sprzętu komputerowego, systemu operacyjnego, warstwy pośredniej (middleware) oraz aplikacji.

Typy klastrów

W punkcie 2. opracowania [1] A. Felkner przedstawia następujące typy klastrów:

— Klastry wysokiej wydajności (*High Performance Computing*) — klastry przetwarzania równoległego, przeznaczone do przetwarzania danych jednego rodzaju. Programy przeznaczone na tego typu klastry wymagają odpowiedniej implementacji, korzystającej z wyspecjalizowanych bibliotek programistycznych.
— Klastry równoważące obciążenie (*Load Balancing Clusters*) — zwane również klastrami serwerowymi. Przeznaczone są do utrzymywania bardzo obciążonych usług sieciowych (serwery pocztowe, WWW, itd.) lub prostych zadań obliczeniowych. Ich działanie polega na równoważeniu obciążenia pomiędzy różnymi serwerami (węzłami klastra). Takie klastry są instalowane w systemach, w których bardzo istotny jest czas reakcji na żąanie klienta.
— Klastry wysokiej dostępności (*High Availibility Clusters*). Klastry tego typu nie zwiększają wydajności, a mają jedynie eliminować pojedynczy punkt awarii (*Single Point of Failure*). W razie uszkodzenia jednego z serwerów, jego zadania są przejmowane przez serwery zapasowe (inne węzły klastra).

W praktyce rozwiązania klastrowe mają charakter mieszany i wykonują dla pewnych aplikacji funkcje wydajnościowe, przy jednoczesnej niezawodności.

2.4.3. Systemy plików używane w klastrach

W rozdziale 3 opracowania A. Felkner opisuje pokrótce systemy plików:

— GFS
— Coda
— NFS
— Lustre
— OpenGFS

OpenGFS oraz Lustre są szczegółowo opisane w pracach Przemysława Danilewicza [3] oraz Łukasza Reszki [4].

Systemy plików w kontekście OpenSSI opisane zostały przez Pawła Kłóska [5]. W klastrze OpenSSI przyjęto założenie, że wszystkie zasoby (łącznie z *devfs* i *procfs*) będą dostępne w formie plików — do których można się dostać przy wykorzystaniu dowolnego komputera w sieci lokalnej. Domyślnie w OpenSSI wykorzystywany jest *Cluster File System* — rozwijany wyłącznie w ramach projektu OpenSSI. Szersze informacje o CFS znajdują się w rozdziale 3 opracowania [5].

Klaster OpenSSI może być jednocześnie klastrem Lustre, można ponadto korzystać z OpenGFS (oraz Sistina GFS)

Zaleca się korzystanie z systemów plików w zależności od zastosowania (wg. P. Kłóska):

— CFS — dla klastrów nie przeznaczonych na przechowywanie danych
— OpenGFS — dla klastrów w których szybkość dostępu do danych jest wartością krytyczną
— Lustre — dla klastrów nastawionych na przechowywanie dużej ilości relatywnie szybko dostępnych danych.

2.4.4. Metody komunikacji międzyprocesowej

Wprowadzenie

Szczegółowa analiza metod komunikacji i synchronizacji międzyprocesowej w OpenSSI przedstawiona jest w pracy Kamila Kołtysia „Realizacja rozproszonej komunikacji międzyprocesowej (IPC) w środowisku OpenSSI" [6]. W rozdziale 1 czytamy:

Komunikacja międzyprocesowa w środowisku OpenSSI realizowana jest poprzez te same obiekty IPC, jakie występują w tradycyjnym systemie Linux. Mamy zatem do dyspozycji:

— potoki
— kolejki FIFO
— sygnały
— kolejki wiadomoąci
— semafory
— pamięć dzieloną
— gniazda (Internet-domain i Unix-domain)

(koniec cytatu)

Kolejki komunikatów — mechanizm ten umożliwia stworzenie kolejki komunikatów, które mogą być odbierane przez inne procesy. Pozwala również na filtrowanie odbieranych komunikatów.

Pamięć dzielona — mechanizm pozwalający na określenie pewnego obszaru pamięci dostępnego dla wszystkich procesów. Koniecznym jest zapewnienie synchronizacji zapisów i odczytów. Ten problem można rozwiązać stosując semafory i przydział konkretnej komórki pamięci dla każdego procesu.

Łącza nienazwane (kolejka FIFO) — umożliwiają jednokierunkowa komunikacje pomiędzy procesami (w ramach jednego programu). Jeden z procesów wysyła dane do łącza, a inny odczytuje w takiej samej kolejności, w jakiej zostały wysłane. Łącze posiada wiec organizację kolejki FIFO (First In First Out). Sama realizacja systemowa opiera się na tworzeniu tymczasowych obiektów w pamięci jądra i udostępnienie ich poprzez interfejs systemu plików. Ponieważ łącza nienazwane są jednokierunkowe (jeden proces zamyka odczyt, drugi zapis) aby zapewnić komunikacje dwustronna trzeba utworzyć dwa kanały dla każdego procesu.

Łącza nazwane — są pewnym rozszerzeniem łączy nienazwanych pozwalającym na komunikacje miedzy dowolnymi procesami (nie tylko w ramach jednego programu). System realizuje to poprzez tworzenie tymczasowych plików (o określonej nazwie) typu FIFO.

Metody komunikacji międzyprocesowej (w kontekście zapewnienia niezawodnych usług) opisane są pokrótce przez K. Ostrowskiego (powyższy opis na podstawie [18], rozdz. 4.3).

IPC w OpenSSI

W rozdziale 1 opracowania Kołtysia [6] czytamy dalej:

Rozproszona komunikacja międzyprocesowa w środowisku OpenSSI jest przezroczysta. Oznacza to, że dla semaforów, kolejek wiadomości i pamięci dzielonej jest jedna przestrzeń nazw w całym klastrze, a potoki, kolejki FIFO i gniazda są wspólne w całym klastrze. Przestrzeń nazw dla IPC jest zarządzana

przez IPC Nameserver, który jest automatycznie reaktywowany po awarii węzła, na którym się on znajdował.

Każdy obiekt IPC jest tworzony na tym węźle, na którym wykonywany jest proces, który go „powołuje do życia”. Ale jest on również dostępny z każdego innego węzła w klastrze, skąd każdy proces może z niego korzystać w dokładnie taki sam sposób, jakby znajdował się on na tym samym węźle, co dany obiekt IPC. Wyjątek stanowią te obiekty IPC, które zostały utworzone przez procesy wykonywane jako lokalne na danej maszynie. Ich zasięg, podobnie jak zasięg procesu jest lokalny i odwoływać się mogą do niego tylko procesy z tego węzła.

Obiekty IPC nie mogą być przenoszone z jednego węzła na inny węzeł, jak np. procesy. Gdy zostaną utworzone na danym weźle, muszą już na nim pozostać aż do momentu ich usunięcia. Dlatego właśnie szybkość działania mechanizmów IPC w środowisku OpenSSI w dużej mierze zależy od tego czy proces będzie migrował, czyli przemieszczał się na inny węzeł czy też nie. Jeżeli bowiem po migracji proces odwoła się do któregoś z obiektów IPC, które wcześniej (przed migracją) utworzył (na innym węźle), to wystąpi opóźnienie związane z koniecznością przesyłania komunikatów przez sieć.

Wydajność mechanizmów komunikacji IPC w OpenSSI

W swoim opracowaniu Kołtyś analizuje wydajność poszczególnych metod komunikacji (semafory, potoki, pamięć dzielona). Jedynie mechanizm semaforów okazuje się być bardzo wydajnym. W przypadku potoków wydajność programu korzystającego z tego mechanizmu IPC zależy od ilości migracji. Dla niektórych problemów (np. procesów wykonujących dużą ilość obliczeń, które można przeprowadzać równolegle) programy oparte o potoki mogą działać wydajnie w środowisku OpenSSI. Pamięć dzielona w tradycyjnym systemie Linux jest najszybszym mechanizmem komunikacji, natomiast w OpenSSI traci ona bardzo wiele ze swojej wydajności. Czym większy jest rozmiar pamięci dzielonej, tym mniej wydajny jest ten mechanizm w OpenSSI. Dlatego trzeba być ostrożnym podczas uruchamiania tym systemie klastrowym programów korzystających z tego mechanizmu komunikacji. (na podstawie rozdz. 5 [6])

System komunikacji międzywęzłowej ICS

ICS jest częścią projektu Cluster Infrastructure (CI), który ma na celu stworzenie powszechnej infrastruktury klastrującej dla rozwiazań opartych na systemie Linux. Za pomocą ICS zrealizowane jest przesyłanie danych na poziomie komunikacji między jądrami węzłów OpenSSI. Dzięki komunikacji ICS klaster OpenSSI jest widziane przezroczyście jako jednen system. ICS zapewnia dwa modele komunikacji: niezawodne przesyłanie wiadomości oraz zdalne wywoływanie procedury.

ICS opisane jest szeroko w opracowaniu M. Remiszewskiego [7]. Czytamy tam:

Na najwyższym poziomie, interfejs pomiędzy ICS a innymi komponentami CI/SSI z niego korzystającymi jest zbiorem usług, z których każda oferuje zestaw dobrze określonych operacji. To właśnie te usługi składają się na interfejs podsystememu ICS.Definiuje się je przy użyciu specjalnego języka, który w czasie kompilacji tłumaczony jest na kod wiążący lokalny komponent z ICS, klienta ICS i serwerem ICS, a w ten sposób komponent ze zdalną usługą. Procesem odpowiedzialnym za generację kodu jest icsgen.

Możliwe jest więc również zrealizowanie na bazie komunikacji ICS własnego systemu rozproszonego.

2.4.5. Clusterproc — pakiet wspomagający zarządzanie procesami w systemach klastrowych

Na podstawie dokumentacji Marcina Pawlaka [6].

Najważniejsze cechy :

— Unikalne w skali klastra numery identyfikacyjne procesów (PID)
— Widoczność i możliwość dostępu do procesów z jakiegokolwiek węzła
— Migracja procesów podczas wykonywania
— Możliwość kontynuacji działania procesu, w sytuacji gdy węzeł macierzysty (na którym proces został stworzony) opuścił klaster.
— Zachowanie integralności, niezależnie od tego, który węzeł uległ awarii
— Relacje parent-child, grupy procesów są rozproszone w klastrze

Identyfikatory procesów

Identyfikatory procesów przydzielane są w następujący sposób: na bitach wysokich kodowany jest numer węzła, natomiast na niskich numer procesu. Identyfikator zgodny jest z typem pid_t. Ilość bitów, na których kodowane są numery węzła określana jest w konfiguracji (domyślnie połowa pid_t).

Procesy zdalne (remote)

Ponieważ mamy do czynienia z rozproszonymi relacjami między procesami tworzone są zastępcze struktury opisujące procesy, wykonujące się na innych węzłach. W strukturze task_struct, za pomocą której w Linuxie opisywany jest proces, została dodana dodana flaga PF_PREMOTE, oznaczająca iż proces znajduje się na zdalnym węźle. Niezbędne informacje o takim procesie przechowywane są w strukturze clusterproc.

Procesy zastępcze nie są szeregowane, nie jest dla nich tworzony stos, nie będą na liście init_task.

System plików /proc

Informacje o systemie udostępnione są się poprzez wirtualny system plików /proc, który jest tworzony i utrzymywany w pamięci przez jądro.

— Readdir będzie przedstawiał wszystkie procesy z wszystkich węzłów oraz inne pliki w /proc
— Readdir dla /proc/node# będzie pokazywał tylko procesy wykonywujące się na określonym węźle.

Relacje parent/child

Pełna lista children/sibling utrzymywana jest na węźle, na którym proces macierzysty jest wykonywany.

Na innych węzłach utrzymywane są informacje o procesach potomnych wykonywanych lokalnie, natomiast w strukturze zastępczej znajduje się informacja o procesie macierzystym.

Relacje między grupami wątków

Grupy wątków prawie zawsze dzielą przestrzeń adresową, w związku z czym członkowie grupy wątków nie są rozdzielani na inne węzły klastra.

2.4.6. Komunikacja sieciowa w OpenSSI

Komunikacja sieciowa w OpenSSI opisana jest w pracy M. Grabowskiego [8].

Zagadnienie komunikacji sieciowej w rozwiązaniu OpenSSI można podzielić na dwie odrębne części:

— komunikację wewnętrzną
— komunikację zewnętrzną

Komunikacja wewnętrzna to niskopoziomowa komunikacja pomiędzy węzłami, natomiast standardowa umożliwia zapewnienie widoczności klastra jako jednej, wysokodostępnej maszyny.

Komunikacja wewnętrzna

Aby umożliwić międzywęzłową komunikację węzeł musi być wyposażony w co najmniej jeden interfejs sieciowy (zwykle ethernetowy). Komunikacja wewnętrzna jest niewidoczna dla świata zewnętrznego. Wszystkie interfejsy komunikacji wewnętrznej klastra muszą być w tej samej podsieci fizycznej. Szczegółowy opis znajduje się w rozdziale 2 opracowania Grabowskiego [8].

Komunikacja zewnętrzna

W założeniu, klaster OpenSSI musi być widoczny z zewnątrz jako jedna, skalowalna i wysoce dostępna maszyna z pojedynczym adresem sieciowym. Jest to bardzo ważne założenie z punktu widzenia sieci. Aby spełnić to założenie w OpenSSI zastosowano rozwiązanie LVS — Linux Virtual Server.

LVS zapewnia, że klaster udostępnia na zewnątrz pojedynczy adres IP, a połączenia przychodzące są rozkładane przy wykorzystaniu rozmaitych algorytmów pomiędzy maszyny obsługujące odpowiednie usługi. Ponadto, równie ważne jest, aby wszystkie urządzenia sieciowe w klastrze (a przez to i adresy IP) wyglądały jak urządzenia lokalne na każdym węźle klastra. Dzięki temu nasłuchiwanie z któregokolwiek węzłów staje się nasłuchiwaniem wszystkich interfejsów sieciowych (zewnętrznych) wszystkich węzłów. Dodatkowo, kompletne rozwiązanie SSI powinno pozwalać na porozumiewanie się dwóch procesów przez interfejs *loopback*, nawet jeśli wykonywane są one na różnych węzłach, a tablice routingowe powinny być synchronizowane tak, że jeśli jakikolwiek węzeł w klastrze może się połączyć z pewnym hostem, wtedy wszystkie węzły powinny mieć taką możliwość. (na podstawie rozdz. 3 [8])

Cluser Virtual IP — wirtualny adres klastra Z opracowania Grabowskiego ([8]):

CVIP to wirtualny adres IP klastra. Adres ten jest używany przez świat zewnętrzny do połączenia z klastrem. Należy zauważyć, że musi on być różny od zestawu adresów IP wewnątrz klastra, tzn. że CVIP nie może być adresem korzenia (ang. *root node*). Plikiem konfguracyjnym zawierającym ustawienia dotyczące CVIP jest /etc/cvip.conf. Jest to plik w formacie XML i zawiera informacje dotyczące CVIP, węzłów głównych (ang. *director nodes*) oraz prawdziwych serwerach (ang. *real servers*).

Węzły główne przekierowują żądania do usług obsługiwanych przez LVS do odpowiednich węzłów zwanych prawdziwymi serwerami, na których dane usługi są faktycznie uruchomione. Węzły główne mogą dzielić funkcjonalność - mogą być zarówno serwerami głównymi, jak i prawdziwymi. W przypadku wielu adresów CVIP albo powinny być one dzielone przez wszystkie węzły główne, albo nie powinny być dzielone przez żadne węzły. Oznacza to, że więcej niż jeden CVIP może być skonfigurowany w klastrze pod warunkiem, że będą miały dokładnie te same zestawy węzłów głównych je obsługujących. Więcej informacji o konfigurowaniu CVIP znajduje się w rozdz. 3.2. [8].

LVS — Linux Virtual Server

Wprowadzenie LVS opisany jest w pracach Jana Boboli [9], oraz Roberta Kuźmiaka [10].

Równoważenie obciążeń w systemie OpenSSI zostało opisane przez Karola Rzońcę w pracy [11] (skupiając się na działanie jądra i migrowania procesów).

Architektura LVS jest trójwarstwowa, składa się z następujących elementów (na podst. rozdz. 1.2 [10]):

— Dzielnik obciążenia (*load balancer*) - jego funkcją jest rozdzielanie połączeń przychodzących od klientów pomiędzy serwery pracujące w klastrze.
— Pula serwerów (*server pool*) - składa się z serwerów, które odostępniają klientom usługi charakteryzujące cały klaster - są to np. ftp, poczta, www, streaming multimediów. Serwery te zwane są również serwerami rzeczywistymi (Real Servers)
— System przechowywania danych (*backend storage*) - najczęściej rozproszony system plików taki jak Koda lub GFS zapewniający spójność danych.

Dzielnik obciążenia (*load balancer*) rozdziela przychodzące połączenia pomiędzy pulę serwerów korzystając z różnych technik dzielenia obciążenia. Jego zadaniem jest również obsługa stanu tych połączeń oraz przekazywanie pakietów do puli serwerów. Cała praca wykonywana przez dzielnik obciążenia jest wykonywana w jego jądrze, co ma pozytywny wpływ na wydajność — LVS jest w stanie obsłużyć dużo więcej połączeń niż zwykły serwer, dzięki czemu nie stanowi wąskiego gardła systemu.

Węzły puli serwerów mogą być replikowane dla potrzeb skalowalności lub wysokiej dostępności.

Wysoka dostępność a skalowalność Skalowalność osiągana jest poprzez przezroczyste dołączanie (odłączanie) węzłów w zależności od obciążenia systemu. Taka wydajność jest (teoretycznie) liniowa, aż do momentu gdy dzielnik obciążenia staje się wąskim gardłem systemu.

Wysoka dostępność jest osiągana niejako przy okazji — w przypadku awarii węzła dzielnik zaprzestaje kierowania do niego połączeń. W przypadku awarii dzielnika — jego funkcję przejmuje zastępca, nasłuchujący „bicia serca” (*heartbeat*) nadrzędnego rozdzielnika.

System składowania danych System składowania danych powinien być redundantym, odpornym na uszkodzenia systemem plików — takim jak Coda czy GFS. Węzły puli serwerów traktują rozproszony system plików jak lokalny system plików. Potencjalnym problemem jest występowanie wyścigów pomiędzy aplikacjami działającymi na różnych węzłach serwera — może być konieczne zastosowanie zarządzcy rozproszonych blokad, który może być częścią rozproszonego systemu plików, albo działać poza systemem plików. (rozdz. 1.2 [9])

Techniki równoważenia obciążenia W LVS stosuje się trzy różne techniki równoważenia obciążenia:

— NAT (*Network Adress Translation*) — tłumaczenie adresów

Równoważenie za pomocą NAT polega na przekierunkowywaniu komunikacji z interfejsu zewnętrznego do wybranej maszyny za z puli serwerów poprzez modyfikację adresów IP oraz odpowiednie rutowanie pakietów. Serwer, do którego komunikacja jest kierowana, wybierany jest zgodnie z algorytmem szeregowania. Sieć puli serwerów jest siecią nierutowalną (prywatną). Technika ta przedstawiona jest na rysunku 1. LVS/NAT wymaga, aby komputery były w jednej klasie sieci lokalnej.

— TUN (*Tunnelling*) — tunelowanie (rysunek 2)

Tunelowanie opiera się o enkapsulację pakietów przez dzielnik obciążenia, a następnie przesyłanie tych pakietów do odpowiedniej maszyny z puli serwerów. Pakiety zwrotne — od serwera — przekazywane są bezpośrednio do klienta, bez udziału dzielnika obciążenia. Serwery rzeczywiste mogę znajdować się w różnych miejscach, niekoniecznie jednej podsieci.

— DR (*Direct Routing*) — rutowanie bezpośrednie (rysunek 3)

Ruting bezpośredni polega na odpowiednim rutowaniu pakietów przez dzielnik obciążenia. Komunikacja do serwera wirtualnego jest rutowana do wybranego serwera rzeczywistego poprzez modyfikację ramek warstwy 2 — adresu MAC. Wszystkie serwery puli serwerów skonfigurowane są na jeden adres wirtualny. Technika ta wymaga, aby serwery rzeczywiste znajdowały się w tej samej podsieci fizycznej.

Techniki równoważenia obciążenia, ich wady i zalety, są szczegółowo opisane w rozdz. 1.2 i 1.3 [10] oraz rozdz. 2 i 4 [9]. Rysunki pochodzą z opracowania Kuźmiaka [10].

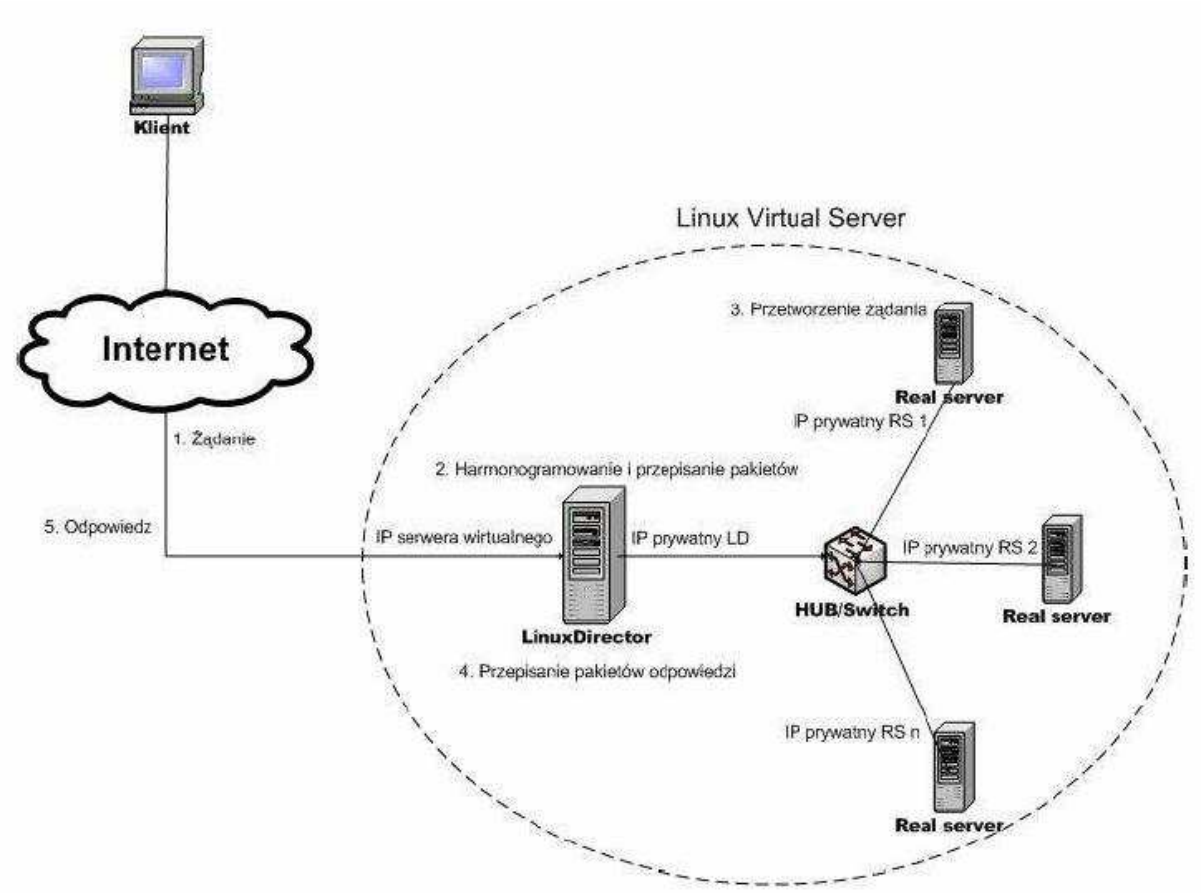
Problem ARP

Zastosowanie tunelowania lub rutingu bezpośredniego (TUN, DR) wiąże się z modyfikacją jądra serwerów rzeczywistych (pracujących pod kontrolą jądra powyżej 2.2.1), tak aby nie odpowiadały na zapytania ARP zewnętrznego IP. Szerzej o tym pisze Kuźmiak w rozdziale 1.3 [10].

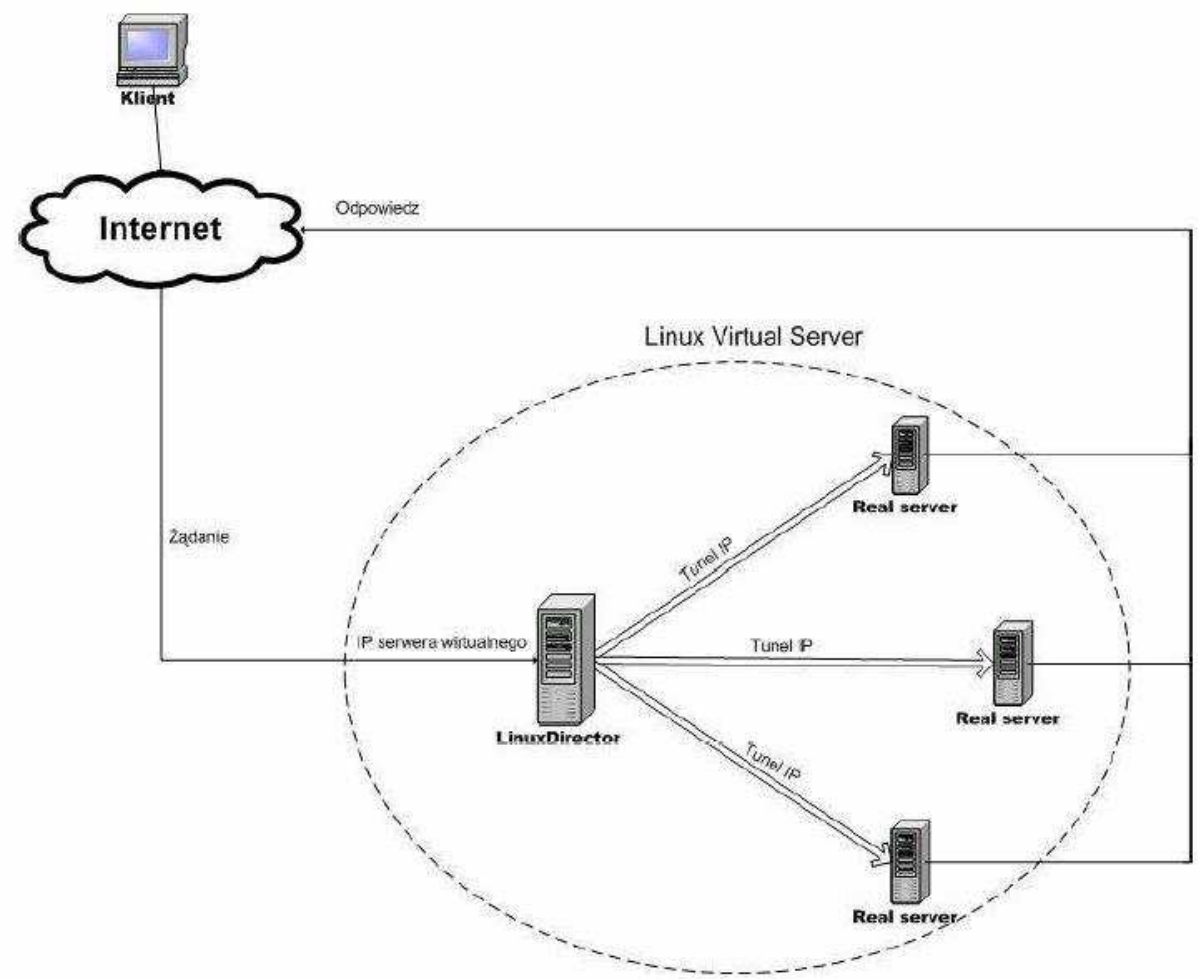
Mając zmodyfikowane jądro należy odpowiednio skonfigurować interfejsy serwerów rzeczywistych (opis w rozdz. 2.5 [10]).

Algorytmy szeregowania W pracach J. Boboli [9] (rozdz. 5) oraz R. Kuźmiaka [10] (rozdz. 1.4) opisane są następujące algorytmy szeregowania:

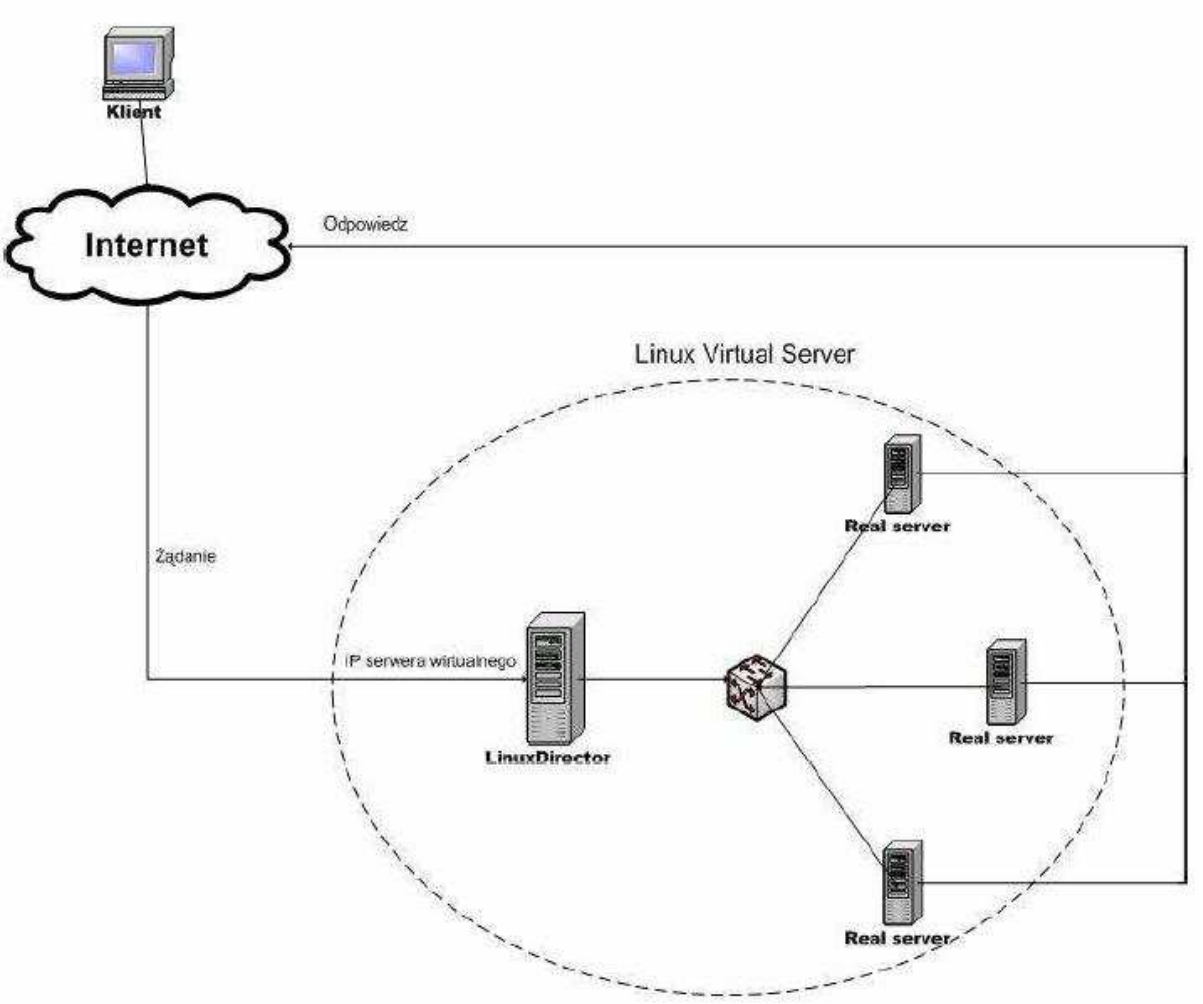
— Round robin (Round-Robin Scheduling)
— Ważony round robin (Weighted Round-Robin Scheduling)
— Najmniej połączeń (Least-Connection Scheduling)
— Ważony najmniej połączeń (Weighted Least-Connection Scheduling)
— Najkrótszy przewidywany czas obsługi (Shortest Processing Time Scheduling)
— Nigdy kolejkowy (Never Queue Scheduling)
— oraz inne: Locality-Based Least-Connection Scheduling, Locality-Based Least- Connection with Replication Scheduling, Destination Hashing Scheduling, Source Hashing Scheduling, Local Node Feature


Klient
Internet
1. Żądanie
5. Odpowiedz
Linux Virtual Server
3. Przetworzenie ządania
Real server
IP prywatny RS 1
2. Harmonogramowanie i przepisanie pakietów
IP serwera wirtualnego
IP prywatny LD
IP prywatny RS 2
HUB/Switch
Real server
LinuxDirector
4. Przepisanie pakietów odpowiedzi
IP prywatny RS n
Real server

Rysunek 2.1.


Klient
Internet
Odpowiedz
Linux Virtual Server
Żądanie
IP serwera wirtualnego
LinuxDirector
Tunel IP
Tunel IP
Tunel IP
Real server
Real server
Real server

Rysunek 2.2.


Klient
Internet
Odpowiedz
Linux Virtual Server
Real server
Żądanie
IP serwera wirtualnego
LinuxDirector
Real server
Real server

Rysunek 2.3.

Zwiększanie niezawodności W tradycyjnym systemie LVS pracuje tylko jeden dzielnik obciążenia, przez co w przypadku jego awarii cały system przestanie działać. Aby zabezpieczyć się przed taką ewentualnością można wprowadzić do systemu zapasowy dzielnik obciążenia, który w stałym okresie czasu będzie sprawdzał czy główny dzielnik pracuje (*heartbeat*). Dodatkowo, główny dzielnik obciążenia może przesyłać do zapasowego informację o aktualnie otwartych połączeniach (zawartość tablicy haszującej), przez co w przypadku awarii głównego dzielnika, połączenia nie zostaną zerwane.

Fake

Fake służy do realizacji niezawodnych usług. W przypadku awarii serwera podstawowego serwer zastępczy przejmuje jego adres IP i zapewnia dalsze działanie usług.

2.4.7. Rozproszone zarządzanie blokadami

Podstawą działania rozproszonego systemu komputerowego jest organizacja dostępu do zasobów. W wyniku rozprzestrzeniania się zasobów, braku globalnej informacji, oraz opóźnień spowodowanych komunikacją, metody synchronizacji stosowane w systemach jednoprocesorowych nie mogą zostać zastosowane w rozproszonych.

Algorytmy oraz pakiety rozproszonego zarządzania blokadami zostały opisane przez M. Majchrowskiego [11]. Omówione zostały algorytmy scentralizowane, algorytmy beztokenowe (algorytm Lamporta), oraz tokenowe (oparte na rozgłaszaniu oraz logicznej strukturze węzłów). Przedstawione zostały także istniejące rozwiązania: OpenDLM (szczegółowo, wraz z opisem instalacji), GULM, SYNCLib, JINI-DLM. P. Trojanek w swoim opracowaniu [13] opisał niedopracowany pakiet LustreDLM.

Open DLM jest w pełni rozproszonym zarządcą blokad. Wiedza o każdej blokadzie jest rozpowszechniana wśród „zainteresowanych” węzłów. Każdy węzeł w takiej sieci posiada i wykorzystuje pełną logikę w celu zezwalania na wejście do sekcji krytycznych. OpenDLM nie opiera się na srukturze scentralizowanej, nie ma żadnego serwera centralnego serwera, a przez to jest odporny na awarie poszczególnych węzłów. (na podst. rozdz. 3.1. [11])

Algorytmy scentralizowane

2.4.8. Realizacja niezawodnych usług w OpenSSI

Problem jest szerzej opisany w pracach Wojciecha Maziarza [16], Marcina Najsa [17], oraz Karola Ostrowskiego [18]. Problem jest najlepiej opisany w opracowaniu Ostrowskiego.

Metody (w skrócie):

— *Keepalive* — na jednym z węzłów działa proces odpytujący usługi działające na innych węzłach. W przypadku, gdy usługa nie odpowiada jest restartowana. Proces demona keepalive jest przypisany do węzła, nie migruje na inne maszyny klastra. Interfejsem linii poleceń jest *spawndaemon*. Aby korzystać z demona keepalive trzeba mieć prawa roota, co jest wadą tego rozwiązania. Szczegółowy opis keepalive znajduje się w opracowaniu W. Maziarza [1].

— OpenSSI nie zapewnia mechanizmu automatycznego odzyskiwania sesji. Istnieje co prawda usługa Kernel Data Replicaction Service, aczkolwiek nie jest ona udokumentowana, ani wykorzystywana w jakimkolwiek z podsystemów (p. 2.2. [18] str. 4 oraz p. 4.3 str. 8 [17]).
— Funkcje udostępnione przez OpenSSI umożliwiające zbudowanie własnej implementacji mechanizmu zapewnienia niezawodności usług opisane są w rozdziale 4 pracy M. Najsa [17] a także w p. 3.2. opracowania K. Ostrowskiego [18]. Ostrowski w swoim opracowaniu proponuje rozwiązanie problemu zapewnienia niezawodnej usługi wraz z utrzymaniem stanu sesji.

Zagadnienia związane z zapewnieniem niezawodności usług (rozdz. 2.1 [18]):

— Restart usługi na innym węźle (komputerze) w klastrze: w przypadku awarii jednego węzła usługa powinna zostać jak najszybciej wznowiona na innym węźle,
— Replikacja danych procesu usługi pomiędzy węzłami: zwiększa odporność klastra na awarie (aż do pełnej replikacji, w której każdy węzeł replikuje swoje sesje do wszystkich pozostałych węzłów, przy takim rozwiązaniu wystarczy, że chociaż jeden z węzłów pozostanie sprawny, aby nie utracić informacji).
— Możliwość rekonfiguracji usługi bez przerywania pracy

2.4.9. Dodatkowe informacje o OpenSSI

Rozproszony dostęp do urządzeń w systemie OpenSSI

Na podstawie dokumentacji Andrzeja Asztemborskiego [15].

Najważniejsze informacje:

— OpenSSI zapewnia przezroczysty dostęp do urządzeń, niezależnie od tego czy znajdują się na lokalnym węźle, czy zdalnym
— Urządzenia odzworowane są w globalnym systemie plików (wpisy w katalogach /dev/# – gdzie # jest numerem węzła)
— Przeniesienie procesu na inny węzeł jest przezroczyste z punktu widzenia dostępu do urzązeń
— Występują problemy z wywołaniem systemowym mount

W dokumentacji opisana jest realizacja przezroczystego dostępu do urządzeń. Przedstawiony jest również przykład dostępu do zdalnego urządzenia.

2.4.10. Literatura

1. Anna Felkner „Porównanie własności różnych dostępnych rozwiązań klastrowych zbudowanych w oparciu o system Linux”
2. Ewa Chachulska „Porównanie własności różnych dostępnych rozwiązań klastrowych zbudowanych w oparciu o system Linux”
3. Przemysław Danilewicz „Rozproszone systemy plików na przykładzie OpenGFS oraz Lustre”
4. Łukasz Reszka „Rozproszone systemy plików na przykładzie OpenGFS oraz Lustre”
5. Paweł Kłósek „OpenGFS Lustre i możliwości ich działania w obrębie klastra OpenSSI”
6. Kamil Kołtyś „Realizacja rozproszonej komunikacji międzyprocesowej (IPC) w środowisku OpenSSI” (ipc_kkoltys.pdf)
7. Maciej Remiszewski „System komunikacji międzywęzłowej ICS” (ics_mremisze.pdf)

8. Michał Grabowski „Komunikacja sieciowa w rozwiązaniu OpenSSI” (ncomm_mgrabowski.pdf)
9. Jan Boboli „Linux Virtual Server” (lvs_jboboli.pdf)
10. Robert Kuźmiak „Linux Virtual Server” (lvs_rkuzmiak.pdf)
11. Karol Rzońca „Zagadnienie równoważenia obciążeń w środowisku OpenSSI” (loadb_krzonca.pdf)
12. Marek Majchrowski „Rozproszone zarzadzanie blokadami” (dlm_mmajchrowski.pdf)
13. Piotr Trojanek „Rozproszone zarzadzanie blokadami” (dlm_ptrojanek.pdf)
14. Marcin Pawlak „Clusterproc — zarządzanie procesami w systemach klastrowych”
15. Andrzej Asztemborski „Dokumentacja dotycząca rozproszonego dostępu do urządzeń w systemie OpenSSI”
16. W. Maziarz „Realizacja niezawodnych usług w rozwiązaniu SSI”
17. M. Najs „Realizacja niezawodnych usług w rozwiązaniu SSI”
18. K. Ostrowski „Realizacja niezawodnych usług w rozwiązaniu SSI”

2.5. Realizacja niezawodności i wydajności poprzez replikację zasobów (P.M)

Jednym z celow budowania systemow rozproszonych jest uczynienie ich bardziej niezawodnymi niz systemy scentralizowane. Niezawodnosc mozemy rozpatrywac w roznych aspektach m.in.: dostepnosci, bezpieczenstwa, odpornosci na bledy. Dostepnosc definiujemy jako czas, w ktorym system jest zdatny do uzytku. Mozemy ja poprawiac minimalizujac ilosc jednoczesnie dzialajacych krytycznych skladowych systemu lub na przyklad stosujac redundancje kluczowych partii sprzetu i oprogramowania, aby w razie potrzeby miec zastepce. Dane przechowywane w systemie musza byc zabezpieczone przed ich utraceniem lub zniekształceniem. Kolejnym aspektem jest odpornosc na bledy, rowniez okreslana jako tolerowanie awarii. Do bledow mozemy zaliczyc wady skladowych systemu. Ogolnie wady klasyfikujemy jako przejsciowe, nieciagle i trwale. Wady przejsciowe nie sa zalezne od samego systemu, a raczej od losowych warunkow zewnetrznych (np. zanik zasilania jednego z serwerow) i samoczynnie ustepuja. Wady nieciagle charakteryzuja sie powtarzajacym sie pojawianiem i samoczynnym znikaniem (np. luzny styk w laczu). Wady trwale natomiast nie ustepuja az do naprawy uszkodzonej skladowej. Awarie skladowych mozemy dodatkowo ropatrywac obserwujac ich zachowanie w stosunku do pozostalych skladowych. Tu sa dwie mozliwosci: uszkodzenie uciszajace i wada bizantyjska. W pierwszym przypadku skladowa po prostu przestaje dzialac (przestaje istniec z punktu widzenia reszty systemu), w drugim - kontynuuje swoje dzialanie robiac wrazenie, ze wszystko jest w porzadku, ale produkuje falszywe sygnaly. Postepowanie z wadami bizantyjskimi jest znacznie trudniejsze od postepowania z uszkodzeniami uciszajacymi. Mozliwe rodzaje awarii to na przyklad awaria laczy, awaria stanowisk, utrata komunikatow. Zapewnienie odpornosci polega na wykrywaniu uszkodzen, rekonfigurowaniu systemu w celu umozliwienia dalszej pracy i przywracaniu stanowisk do pracy po awarii. Kolejne fazy przedstawiaja sie nastepujaco:

— Wykrywanie uszkodzen
  – wykrywamy, ze wystapilo jakies uszkodzenie, ale nie wiemy konretnie jakie, gdyz z reguly nie mozna okreslic jego rodzaju (ciezko odroznic awarie lacza, awarie stanowiska i utrate komunikatu); czasem jest to mozliwe, jesli przykladowo miedzy stanowiskami A i B istnieje jakies alternatywne polaczenie (mozna wtedy sprawdzic, czy zawiodlo polaczenie, czy stanowisko)
  – w celach wykrywania uszkodzen stosujemy na przyklad procedure uzgadniania (*ang. handshaking*)

    Procedura uzgadniania:

    Zalozmy, ze stanowiska A i B maja bezposrednie, fizyczne polaczenie. W stalych odstepach czasu oba

stanowiska wysylaja sobie wzajemnie komunikat „jestem czynne”. Jesli stanowisko A nie otrzyma tego komunikatu w okreslonym czasie (*timeout*), to moze przyjac, ze stanowisko B ulego awarii, ze popsulo sie lacze A-B lub ze komunikat z B zostal zgubiony. W tej sytuacji A moze postapic dwojako. Moze przeczekac prze kolejny okres na otrzymanie od B komunikatu „jestem czynne” albo tez moze wyslac do B komunikat „czy jestes czynne ?”. Jesli stanowisko A nie otrzyma komunikatu „jestem czynne” ani odpowiedzi na wyslane przez nie w tej sprawie zapytanie, to procedura moze zostac powtorzona. Jedyny wniosek, jaki mozna bezpiecznie wyciagnac na stanowisku A to ten, ze wystapila jakas awaria. Stanowisko A moze probowac odroznic awarie lacza od awarii stanowiska, jesli istnieje jakies alternatywne polaczenie do B.

— Rekonfigurowanie
  – jesli wiadomo, ze uszkodzone jest polaczenie miedzy A i B, to poinformuj o tym reszte stanowisk (np. w celu uaktualnienia tablic tras)
  – jesli wiadomo, ze uszkodzone jest stanowisko X, to poinformuj o tym pozostale stanowiska (jesli to bylo stanowisko koordynatora, to wymagane bedzie wybranie nowego)
— Przywracanie stanowiska do pracy po awarii
  – przywrocenie lacza - poinformuj o tym wszystkie stanowiska
  – przywrocenie stanowiska - poinformuj o tym wszystkie stanowiska i uaktualnij dane na przywracanym stanowisku

Walczyc z awariami (neutralizowac czulosc na nie) mozemy stosujac redundancje. Rozrozniamy trzy rodzaje redundancji: redundancja informacji, redundancja czasu i redundancja fizyczna. Redundancje informacji tworza dodatkowe bity umozliwiajace odtwarzanie zniekształconych bitow (np. kod Hamminga stosowany w celu zniwelowania wplywu szumow linii transmisyjnej). Redundancja czasu polega na wykonaniu jakiegos dzialania i — w razie potrzeby — na jego powtorzeniu. Ten rodzaj redundancji jest szczegolnie pomocny tam, gdzie w gre wchodza wady przejsciowe lub nieciagle. Ostatnia, redundancja fizyczna, powstaje przez dodanie specjalnego wyposazenia, aby system jako calosc mogl tolerowac utrate pewnych skladowych lub ich wadliwe dzialanie. Pojawia sie tu jednak problem zachowania spojnosci kopii. Redundancje fizyczna dodatkowo dzielimy na zwielokrotnianie aktywne i pasywne. Odmiana aktywna charakteryzuje sie tym, ze kazde zapytanie/transakcja wykonywane sa na wszystkich replikach. Zaleta takiego podejscia moze byc zwiekszenie wydajnosci (o czym bedzie jeszcze dalej). O zwielokrotnianiu pasywnym mowimy wtedy, kiedy zapytanie/transakcja wykonywane sa na jednej tylko replice. Pozostale repliki sluza za rezerwe (oczywiscie musze byc w miedzy czasie uaktualniane) - jesli oryginalna kopia zginie, jest zastepowana przez ktoras z replik. W porownaniu ze zwielokrotnianiem aktywnym mozna tu zauwazyc pewna zalete - jest ono latwiejsze w czasie normalnego dzialania systemu, gdyz komunikaty podazaja tylko do jednego serwera, a nie do calej grupy. Wada jest zle dzialanie w przypadku wystepowania wad biznatyjskich i dluzszy czas przywracania stanowiska podstawowego do pracy po awarii.

Innym celem (nie zawsze jednak osiaganym) stosowania systemow rozproszonych jest uzyskanie lepszej niz w przypadku scentralizowanym wydajnosci. Wydajnosc moze byc oceniana na przyklad poprzez pomiar czasu realizacji transakcji/zapytan w systemie. Jednym ze sposobow osiagania lepszej wydajnosci jest wspomniane juz stosowanie zwielokrotniania aktywnego. Wyrozniamy trzy rodzaje tego typu replikacji: jawne, leniwe i grupowe. Zwielokrotnianie jawne pozostawia w kwestii programisty tworzenie replik zasobu. To on mowi co gdzie ma byc powielone. Zwielokrotnianie leniwe polega na tworzeniu tylko jednej kopii zasobu na pewnym serwerze. W pozniejszym czasie sam serwer dokonuje nastepnych zwielokrotnien automatycznie, bez wiedzy programisty. wymaga to od systemu poswiecaniu szczegolnej uwagi spojnosci kopii i w razie potrzeby odzyskania, wskazania wlasciwej kopii. W zwielo-

krotnianiu grupowym operacja modyfikowania powielonego zasobu jest jednoczesnie realizowana na wszystkich serwerach.

Atualizowanie zwielokrotnionego zasobu moze byc realizowane na rozne sposoby. Wymienie tu trzy. Pierwszym sposobem jest zwielokrotnianie kopii podstawowej. Przy jego zastosowaniu jeden z serwerow pelni role podstawowego (*master*). Wszystkie inne sa wtorne (*slave*). Wszelkie zmiany wprowadza sie na oryginale (w masterze), a master powiadamia wszystkie slave'y o tym, co trzeba zaktualizowac. Aby ustrzec sie przed skutkami awarii serwera podstawowego, nastepujacej przed poinstuowaniem przez niego wszystkich serwerow wtornych, aktualizacja powinna zostac zapisana w pamieci trwalej jeszcze przed zmiana kopii podstawowej. Dzieki temu po wznowieniu przez serwer pracy po awarii mozna dokonac sprawdzenia, czy jakies aktualizacje byly w toku w chwili jej wystapienia. Jesli tak, to mozna je odtworzyc. Metoda ta jednak ma wade — jesli serwer podstawowy zostanie wylaczony, to nie mozna wykonac zadnej aktualizacji. Innym sposobem, radzacym sobie z tym problemem, jest algorytm glosowania (*voting*). Podstawowy pomysl polega na wymaganiu od klientow, aby przed proba odczytu lub zapisu powielonego zasobu uzyskiwali pozwolenia od wielu serwerow.

Oto ilustracja dzialania tego algorytmu:
Zalozmy, ze plik jest powielony na N serwerach. Mozemy ustanowic regule, ze w celu zaktualizowania pliku klient musi najpierw skontaktowac sie przynajmniej z polowa serwerow plus 1 (wiekszosc) i uzyskac od nich zgode na wykonanie aktualizacji. Po ich zgodzie plik zostaje zmieniony, a jego nowa wesja zostaje opatrzona nowym numerem. Numer wersji jest uzywany do identyfikowania wersji pliku i jest taki sam dla wszystkich nowo zaktualizowanych plikow. W celu przeczytania zwielokrotnionego pliku klient musi rowniez skontaktowac sie z przynajmniej polowa serwerow plus 1 i poprosic je o wyslanie skojarzonych z plikiem numerow wersji. Jesli wszystkie numery wersji beda zgodne, to musi to byc wersja najnowsza, gdyz proba zaktualizowania reszty serwerow musialaby sie zakonczyc niepowodzeniem, bo byloby ich za malo.

Ogolny algorytm mozna opisac nastepujaco:

— do czytania pliku majacego N zwielokrotnien wymaga sie od klienta zgromadzenia *quorum czytana*, czyli dowolnego zbioru Nc lub wiecej serwerow
— do zmodyfikowania tego pliku wymaga sie od klienta zgromadzenia *quorum pisania*, czyli dowolnego zbioru Np lub wiecej serwerow
— musi zachodzic: $Nc + Np > N$

Ostatnim algorytmem jest modyfikacja glosowania — glosowanie z martwymi duszami (*voting with ghosts*). Jest to rozszerzenie algorytmu o radzenie sobie z problemem, kiedy usterce ulegnie taka liczba serwerow, ktora nie pozwoli uzyskac quorum pisania (zazwyczaj Np jest bliskie N, ze wzglegu na proporcje wystepowania operacji zapisu i czytania). Zastepujemy wowczas kazdy serwer, ktory ulegl awarii, serwerem pustym - martwa dusza. Serwer taki nie moze uczestniczyc w quorum czytania (bo nic na nim nie ma), ale moze uczestniczyc w quorum pisania (ignorujac wszelkie kierowane do niego operacje zapisu). Pisanie konczy sie sukcesem tylko wtedy, gdy przynajmniej jeden serwer istnieje naprawde.

W algorytmach, w ktorych jeden z serwerow jest wyrozniony jako koordynator (np. w algorytmie zwielokrotniania kopii podstawowej) jest niebezpieczenstwo, ze wyrozniona maszyna ulegnie awarii. W takich przypadkach mozemy wspomoc sie algorytmami elekcji, ktore pomoga w ustaleniu nowego koordynatora. Algorytmami elekcji sa np. algorytm tyrana i algorytm pierscieniowy.

3. Faza II

3.1. Projekt wstępny rozwiązania końcowego (T.Ł.)

Do realizacji zadania projektowego zamierzamy wykorzystać mechanizm replikacji, który jest już zaimplementowany w systemie MySQL. Mamy zamiar zmodyfikować go w sposób, który pozwoli uzyskać znaczny wzrost wydajności dla operacji SELECT, a jednocześnie uzyskać bardzo wysoką odporność na uszkodzenia.

3.1.1. Architektura rozwiązania

Kompletny system będzie działał na określonej liczbie węzłów (na starcie; w trakcie pracy liczba ta może się to zmieniać). Na Każdym z wezłów będzie uruchomiona jedna instancja systemu RDBMS MySQL. Wszystkie działające w danej chwili instancje będą zorganizowane w odpowiednią hierarchię replikacji (nie jest ona jeszcze ustalona), z wydzielonym węzłem głównym.

Zapytania od klientów będą kierowane do węzłów podrzędnych. Po przeanalizowaniu zapytania węzeł podejmie decyzję o tym kto ma wykonać to zapytanie. Zapytania, które nie modyfikują zawartości bazy danych, będą wykonywane przez węzły podrzędne przy użyciu lokalnej repliki. Pozostałe zapytania będą kierowane do węzła głównego. Propagację zmian dokonanych przez węzeł główny zapewni mechanizm replikacji bazy danych MySQL.

3.1.2. Wydajność systemu

Przyjęte rozwiązanie pozwoli uzyskać tak wysoką wydajność (mierzoną liczbą obsłużonych zapytań w jednostce czasu) zapytań SELECT, jak to jest tylko możliwe. Dla pozostałych zapytań ich liczba możliwa do wykonania w jednostce czasu będzie zbliżona do bazy danych działającej na pojedynczym węźle, choć czas odpowiedzi na zapytanie będzie dłuższy.

3.1.3. Odporność na uszkodzenia

Możliwe jest takie zaimplementowanie tego rozwiązania, aby system świadczył swoje usługi, nawet po degradacji do tylko jednego sprawnego węzła. Najtrudniejszą sytuacją, z jaką system będzie musiał sobie poradzić, będzie awaria węzła głównego — konieczne będzie zaimplementowanie dynamicznych zmian w hierarchii replikacji, łącznie z wybieraniem nowego węzła głównego.

3.1.4. Wykorzystanie mechanizmów OpenSSI

Mamy zamiar wykorzystać mechanizmy udostępnianie przez platformę OpenSSI (głównie mechanizmy komunkacji międzyprocesowej) do realizacji następujących obszarów funkcjonalności:

— równoważenie obciążenia węzłów podrzędnych przez kierowanie zapytań klientów do najmniej obciążonych węzłów,
— śledzenie stanu innych replik,
— komunikacja między replikami w sytuacji awarii jednego węzła lub wielu węzłów.

4. Faza III

4.1. Testowanie wydajności (T.M. i W.S.)

4.1.1. Testy

Do wykonania testów wybraliśmy narzędzie *Super Smack* [1]. Jest to często stosowany pakiet służący do testowania wydajności, odporności na obciążenia i generowania dużych obciążeń w bazach MySQL.

Super Smack dysponuje możliwością wykonywania dużych ilości zapytań `select` oraz `update`, które całkiem dobrze oddają charakterystykę typowych odwołań do bazy będącej zapleczem dla witryny internetowej. [2]

W testach skorzystaliśmy z przykładowych konfiguracji dostarczanych razem z całym pakietem. Należało jedynie uzgodnić prawa dostępu oraz adresy serwerów.

4.1.2. Wersja klastrowa

Niniejsze testy przeprowadziliśmy na wersji *5.1.7-beta* bazy MySQL, czyli tej, która stanowiła zasadniczą część drugiego etapu projektu. Testy zostały wykonane na bazie działającej na trzech węzłach klastra *OpenONE*.

Pola tabeli zawierają średni czas połączenia (w milisekundach) i ilość wykonanych operacji na sekundę.

Select

ilość zapytań	liczba łączących się klientów				
	1	5	10	15	20
		63	724	966	1561
1000		5547.92	5326.25	5513.82	5673.65
	0		731		
10000	3513.12		5608.07		

[1] http://vegan.net/tony/supersmack

[2] Tony Bourke. „Using MySQL to benchmark OS performance”, luty 2005

Update

ilość zapytań	liczba łączących się klientów				
	1	5	10	15	20
		1	42	80	86
1000		182.70	510.60	363.73	189.25
	1		31		
10000	463.72		461.88		

4.2. Loadbalancing (K.R.)

4.2.1. Problemy z LVS w OpenSSI

Uruchomienie loadballancingu do obsługi połączeń sieciowych, nie powiodło się. Nasłuchując na dowolnym z poniższych adresów:

```
* openone (194.29.167.57)
* cvip (194.29.167.57)
* default (0.0.0.0)
* node * IP (10.0.0.*) na ustalonym nodzie
```

Połączenia do CVIP nie były odbierane poprawnie. Jedyny przypadek, gdy połączenia były przyjmowane na CVIP to sytuacja, gdy proces nasłuchujący uruchomiony jest na węźle directora (węźle 1.).

```
# Przykład (brak połączenia)
onnode 3 nc -l 10.0.0.3 -p 4567
onnode 1 nc 194.29.167.57 4567

# Przykład (brak połączenia)
onnode 2 nc -l -p 4567
onnode 1 nc 194.29.167.57 4567

# Przykład (brak połączenia)
nc -l -p 4567
onnode 1 nc 194.29.167.57 4567

# Przykład (brak połączenia, w przypadku gdy nasłuchiwanie nie zostanie rozpoczęte na wę
nc -l -p 4567
nc openone 4567

# Przykład (połączenie)
onnode 1 nc -l -p 4567
nc 194.29.167.57 4567
```

Bez praw administratora – braku dostępu do skryptów administracyjnych (np /sbin/ipvsadm -L) a także możliwości „poeksperymentowania” na konfiguracji bardzo trudnym jest znalezienie przyczyny powyższych problemów i zaproponowanie ich rozwiązania. W związku z tym integracja serwera MySQL w

klastrze, wykorzystująca właściwości OpenSSI nie jest możliwa. Koncepcja zastępcza W związku z niepoprawnym działaniem dzielnika obciążeń OpenSSI byliśmy zmuszeni zaproponować własne rozwiązanie: serwer proxy, przekierowujący połączenia do najmniej obciążonego węzła, na podstawie tablicy systemowej /proc. Serwer ten jest przywiązany do węzła 1, w momencie nadejścia połączenia otwiera połączenie do wybranego adresu IP węzłów (10.0.0.*) i w przypadku poprawnego połączenia kopiuje przychodzące dane pomiędzy gniazdami. Wadą tego rozwiązania jest wysoka warstwa obsługi przechodzących danych – nie ma możliwości operowania na buforach systemowych (warstwy 3 stosu TCP/IP) i w związku z tym tunelowanie dla dużych danyc nie jest zbyt wydajne. Tak więc obsługa żadań SQL zwracających duże ilości danych jest w tej sytuacji mało efektywna. W bazach transakcyjnych, jednakże, nie jest to częsty przypadek – ilość przesyłanych danych jest raczej ograniczona, więc można założyć, że będzie to dopuszczalne rozwiązanie dla testowania wydajności. Aby zapewnić dużą ilość równoległych połączeń należałoby zrealizować serwer proxy na wątkach.

4.3. Projekt rozwiązania końcowego (T.Ł.)

4.3.1. Wprowadzenie

Można za to zauważyć, że implementacja standardowego klastra MySQL pozostawia wiele do życzenia w sferze odporności na uszkodzenia:

— klaster nie jest w stanie przetrwać awarii węzła, na którym został uruchomiony serwer zarządzający ndb_mgmd,
— zmiana konfiguracji klastra (np. dodanie nowego węzła lub powtórne przyłączenie węzła po jego awarii) nie jest możliwa bez wyłączenia całego systemu (czyli wymagana jest przerwa w pracy klastra)

Po analizie możliwości klastra MySQL doszliśmy do wniosku, że zaprojektowanie i implementacja rozwiązania o lepszej odporności na uszkodzenia leży w zasięgu naszych możliwości. Oczywiście rozwiązanie takie będzie miało również odpowiednią wydajność dzięki rozproszeniu przetwarzania zapytań SQL na wiele węzłów klastra OpenSSI.

4.3.2. Replikacja

U podstaw wszelkich rozwiązań klastrowych baz danych leży mechanizm replikacji. Obecna wersja MySQL daje możliwość skorzystania z dwóch implementacji tego mechanizmu:

1. *MySQL cluster* opiera się na synchronicznej replikacji wbudowanej w mechanizm składowania danych ndb.
2. Dostępna jest asynchroniczna, jednokierunkowa replikacja *master – slave*, włączana i wyłączana przez administratora bazy.

Drugie z tych rozwiązań ma pewną właściwość, bardzo cenną przy realizacji rozwiązań o podwyższonej odporności na uszkodzenia: konfiguracja mechanizmu replikacji (czyli ustalenie relacji master – slave, ustalenie stanu początkowego bazy danych dla węzła podrzędnego oraz uruchomienie mechanizmu replikacji) jest możliwa bez przerywania pracy węzła nadrzędnego (jeśli korzystamy z mechanizmu składowania *InnoDB*, można skorzystać z narzędzia *Hot Backup*, które umożliwia utworzenie repliki

nawet bez chwilowego blokowania zapisu do bazy w węźle nadrzędnym). Dzięki temu możliwe jest uruchomienie systemu bazodanowego o zmiennej w czasie hierarchii węzłów pozostających między sobą w relacjach master – slave, a więc dynamicznie rekonfigurującego się. Przy zastosowaniu odpowiedniego protokołu komunikacji i uzgadniania stanu całego systemu przez poszczególne węzły możliwe byłoby uzyskanie rozwiązania o bardzo dużej odporności na uszkodzenia — byłby on w stanie działać nawet po degradacji do pojedynczego węzła, a także wrócić do stanu pełnej sprawności po usunięciu przyczyny awarii, cały czas świadcząc usługi.

Zdecydowaliśmy, że poszczególne repliki bazy danych będą łączone w strukturę drzewa binarnego. Każda replika (oczywiście poza nadrzędną, czyli korzeniem drzewa) będzie miała swój węzeł nadrzędny (*master*), sama będąc jednocześnie nadrzędną repliką dla maksymalnie dwóch innych węzłów. Możliwe są oczywiście inne rozwiązania, od struktury prawie całkowicie płaskiej (jeden węzeł nadrzędny, wszystkie pozostałe są jego bezpośrednimi replikami) do liniowej. Drzewo binarne wydaje się być dobrym kompromisem, gdyż nie zachodzą tu przypadki skrajne: żaden węzeł nie jest obciążony kosztami przesyłania informacji o wszystkich zmianach do wszystkich pozostałych, opóźnienie w propagacji zmian nie jest również zbyt duże.

4.3.3. Architektura systemu

Projektowana przez nas klastrowa wersja systemu MySQL składa się z trzech zasadniczych komponentów.

Serwer rozpraszający zapytania

Jednym z wymagań postawionych w treści zadania projektowego jest przezroczystość klastrowego rozszerzenia bazy danych — programy klienckie MySQL mają łączyć się do klastra w sposób identyczny jak do standardowego serwera MySQL. Środowisko OpenSSI udostępnia bardzo dobre narzędzie do realizacji tego celu: *Linux Virtual Server*. W najprostszej wersji jest to rozszerzenie linuksowego filtra pakietów, wykonujące translację adresów IP w taki sposób, aby połączenia z sieci zewnętrznej były kierowane do różnych węzłów sieci wewnętrznej. LVS w środowisku OpenSSI wykorzystuje mechanizmy monitorowania obciążenia węzłów, aby kierować przychodzące połączenia do najmniej obciążonych węzłów. Niestety, konfiguracja LVS na dostępnej w laboratorium instalacji OpenSSI nie przewiduje możliwości łączenia się do klastra z zewnątrz.

Ostatecznie zdecydowaliśmy się na zaimplementowanie prostego serwera pośredniczącego w przesyłaniu danych pomiędzy klientami MySQL a węzłami klastra. Program taki jest typowym serwerem współbieżnym: oczekuje na połączenia TCP na blokującym wywołaniu `accept()`, a dla każdego nowego połączenia powołuje proces potomny, który od tej pory jest odpowiedzialny za przesyłanie danych. Proces potomny pobiera z systemu listę aktywnych węzłów wraz z informacją o ich obciążeniu (w sposób analogiczny jak polecenie `loads`), nawiązuje połączenie z serwerem MySQL na węźle o najniższym obciążeniu, a następnie aż do zakończenia połączenia przekazuje dane pomiędzy klientem a serwerem MySQL korzystając z funkcji systemowej `select()`.

Taka realizacja rozpraszania zapytań do węzłów bazy danych ma dwie wady w porównaniu z rozwiązaniem opartym na LVS:

1. LVS jest realizowany w warstwie trzeciej modelu ISO, tzn. przetwarzane są jedynie nagłówki pakietów IP, a same pakiety przekazywane są pomiędzy kartami sieciowymi a buforem jądra systemu.

Realizacja rozpraszania zapytań w przestrzeni użytkownika powoduje, że każdy przychodzący pakiet jest dodatkowo przetwarzany przez warstwę TCP, kopiowany do przestrzeni adresowej procesu, a następnie znów przekazywany „w dół” stosu protokołów TCP/IP. Oczywiście powoduje to dodatkowe obciążenie węzła, na którym ten serwer działa.

2. OpenSSI zawiera specyficzną implementację LVS: połączenia są rozdzielane tylko do tych węzłów, na których istnieje proces oczekujący na połączenia na zadanym porcie — taki, który wywołał funkcję `listen()` dla gniazda przypisanego do właściwego portu. W naszym rozwiązaniu zakładamy, że serwer MySQL jest uruchomiony na stałej grupie węzłów klastra (opisanej w odpowiednim pliku konfiguracyjnym), a sytuacja, w której pewien węzeł działa, ale nie ma na nim procesu serwera MySQL, jest sytuacją wyjątkową. Może się tak stać po wystąpieniu zaburzeń w pracy klastra (np. awarii sieci lub węzła), kiedy klaster MySQL jest w trakcie rekonfiguracji — w takiej sytuacji pewne połączenia od klientów MySQL mogą zostać odrzucone.

Serwer pośredniczący powinien być uruchomiony na węźle numer 1 klastra OpenSSI — jego obecność jest wymagana przez cały czas działania klastra OpenSSI, nie istnieje więc ryzyko utraty możliwości podłączenia do bazy danych przez awarię węzła.

Serwer bazy danych MySQL

Wybrany przez nas mechanizm asynchronicznej, jednokierunkowej replikacji danych jest już obecny w kodzie MySQL. Przyjęta metoda replikacji nakłada na nas konieczność wprowadzenia pewnej zmiany w kodzie odpowiedzialnym za przetwarzanie zapytań SQL.

Propagacja informacji o zmianach w bazie danych odbywa się tylko w jednym kierunku, od węzła nadrzędnego do węzłów podrzędnych. Naturalną konsekwencją tego jest konieczność szczególnego traktowania wszelkich zapytań modyfikujących zawartość bazy — aby informacje o zmianach trafiły do wszystkich węzłów, muszą one pochodzić z węzła stojącego najwyżej w hierarchii. Oznacza to, że tylko jeden węzeł może przetwarzać zapytania, które modyfikują zawartość bazy danych — wszystkie inne mogą być przetwarzane przez dowolny z węzłów.

Rozważaliśmy dwa możliwe podejścia do tego problemu: umieszczenie kodu rozpoznającego „krytyczne” zapytania w serwerze rozpraszającym zapytania (co na pierwszy rzut oka wydaje się być naturalne) oraz rozpoznawanie zapytań w węzłach. Z pierwszej możliwości zrezygnowaliśmy, gdyż oznaczałaby ona dwukrotne wykonywanie znaczącej części pracy (interpretacji SQL) przy każdym zapytaniu. Wykonywanie tego w węzłach bazy wprowadzi dużo mniejszy narzut na wydajność całego systemu — najczęściej powtarzające się zapytania typu **S**ELECT będą interpretowane tylko raz.

Ostatecznie zdecydowaliśmy się na najprostszy z możliwych wariant modyfikacji kodu serwera MySQL. Jeżeli zapytanie zostanie zidentyfikowane jako modyfikujące zawartość bazy, proces odpowiedzialny za obsługę danego klienta przejdzie nieodwracalnie w tryb przekazywania zapytań: każde kolejne zapytanie od klienta zostanie przesłane do węzła będącego aktualnie na szczycie hierarchii (jeżeli takiego nie ma, zapytanie zostanie odrzucone), a wyniki będą przekazywane klientowi. Szczególnym przypadkiem jest sytuacja, kiedy zapytanie modyfikujące bazę jest elementem transakcji (innej niż transakcja obejmująca pojedyncze zapytanie w standardowym trybie pracy *autocommit*) — węzeł obsługujący klienta po połączeniu się z węzłem nadrzędnym musi jako pierwsze zapytanie przekazać BEGIN TRANSACTION. Takie postępowanie gwarantuje zachowanie poziomu spójności *read committed*.

Program zarządzający hierarchią klastra

Program zarządzający jest kluczowym elementem systemu, koordynującym współpracę replik serwera MySQL działających na poszczególnych węzłach klastra OpenSSI. Na każdym węźle, na którym przewidziane jest uruchamianie jednej z replik serwera, musi być przez cały czas uruchomiony program zarządzający. Pełni on dwie podstawowe funkcje.

1. W czasie normalnej pracy systemu procesy działające na poszczególnych węzłach nieustannie komunikują się ze sobą, wymieniając informacje o stanie całego systemu. Dzięki temu możliwe jest wykrycie sytuacji wyjątkowej (takiej jak awaria sieci lub jednego z węzłów) i podjęcie odpowiedniej reakcji.
2. Na starcie systemu lub po awarii konieczne jest ustalenie hierarchii replik, tzn. ustalenie relacji master – slave pomiędzy poszczególnymi replikami bazy danych. W tym celu instancje programu zarządzającego próbują „odszukać" inne węzły wysyłając zapytania na z góry określony port przy użyciu bezpołączeniowego protokołu UDP, a następnie dołączyć je do już istniejącej hierarchii. Dokładniejszy opis takiego działania znajduje się poniżej.

Dodatkowo, program zarządzający zapisuje do logów systemowych informacje diagnostyczne o stanie systemu, aby możliwe było śledzenie wszelkich zmian i podejmowanie interwencji w sytuacjach takich jak awarie sprzętu.

4.3.4. Działanie procesów zarządzających

W trakcie normalnej pracy programy zarządzające poszczególnymi replikami zajmują się jedynie monitorowaniem stanu systemu. Każdy węzeł jest korzeniem pewnego poddrzewa i w określonych odstępach czasu wysyła do swojego węzła nadrzędnego stan własnego poddrzewa. W ten sposób korzeń całego drzewa przez cały czas dysponuje aktualną wiedzą o stanie całości. Taka informacja (mapa całego drzewa) jest w regularnych odstępach czasu przesyłana w dół drzewa. Towarzyszy jej dodatkowe pole, określające który z węzłów jest w danym momencie nadrzędnym z punktu widzenia replikacji danych, tzn. do którego z węzłów należy przekazywać zapytania SQL modyfikujące bazę danych. Numer tego węzła może mieć wartość -1, co jest informacją dla całego poddrzewa, że węzeł nadrzędny nie jest dostępny, a więc ta część drzewa replikacji nie może dalej świadczyć usług.

Wyzwaniem dla procesu zarządzającego repliką jest sytuacja, kiedy informacje o stanie całego drzewa procesów nie są wymieniane, tzn. kiedy z jakiegoś powodu wystapią problemy z komunikacją danego węzła z pozostałymi oraz na starcie systemu. Poniżej zostaną omówione te dwa przypadki.

Start systemu

Podczas uruchamiania systemu jeden z węzłów jest wyróżniony spośród pozostałych. Jest to pierwszy węzeł nadrzędny — ten, który rozpoczyna pracę z przygotowaną wcześniej kopią bazy danych. Proces zarządzający repliką po uruchomieniu jest korzeniem własnego poddrzewa (i oczywiście jedynym jego węzłem). Rozpoczęcie działania polega na uruchomieniu procedury łączenia drzew (opisanej poniżej).

Awaria

Za awarię uważamy sytuację, w której proces zarządzający repliką przez określony (definiowany przez administratora) czas nie otrzyma informacji od swojego węzła nadrzędnego lub jednego ze swoich węzłów podrzędnych. Postępowanie w obu przypadkach jest różne.

W przypadku utraty połączenia z węzłem podrzędnym proces zarządzający przesyła do swojego węzła nadrzędnego nową mapę własnego poddrzewa, z usuniętym węzłem, z którym właśnie utracił połączenie (oraz, co zrozumiałe, jego węzłami podrzędnymi). Informacja ta dotrze do korzenia drzewa (w czasie normalnej pracy będzie on również nadrzędnym węzłem replikacji), który będzie musiał sprawdzić, czy jest możliwe kontunuowanie świadczenia usług, tzn. czy liczba węzłów, które pozostały w drzewie, jest większa niż połowa liczby węzłów przy poprzednim stanie drzewa. Jeżeli warunek ten nie jest spełniony, korzeń drzewa przestanie być węzłem nadrzędnym replikacji. Następnie korzeń prześle w dół drzewa nową jego mapę (być może z informacją, że należy zaprzestać świadczenia usług).

Utrata połączenia z węzłem nadrzędnym oznacza, że proces, który taką sytuację wykrył, stał się korzeniem własnego drzewa. Musi on sprawdzić warunek przytoczony w poprzednim akapicie — jeżeli liczność jego własnego drzewa jest większa, niż połowa liczby węzłów drzewa w poprzednim stanie, to staje się on nowym węzłem nadrzędnym replikacji. Niezależnie od tego warunku, musi on przesłać w dół drzewa jego nową mapę.

Łączenie drzew

Postępowanie według opisu zawartego w poprzednich paragrafach prowadzi do sytuacji, kiedy zamiast jednego drzewa, obejmującego wszystkie działające w danym momencie węzły, istnieje wiele mniejszych drzew. Pożądanym zachowaniem jest oczywiście połączenie takich drzew w jedno — rozbicie powoduje, że jedynie węzły należące do drzewa, którego korzeniem jest węzeł nadrzędny replikacji, dalej świadczą usługi (oczywiście, nie zawsze takie drzewo będzie istniało).

Węzeł, który jest korzeniem drzewa o liczności mniejszej niż liczba wszystkich węzłów systemu (konfigurowana przez administratora na starcie) przez cały czas aktywnie próbuje nawiązać komunikację ze wszystkimi węzłami, które aktualnie nie są elementami jego drzewa. W akapicie opisującym start systemu zostało to określone jako procedura łączenia drzew.

Jeżeli węzłom, które nie są elementami jednego drzewa, uda się rozpocząć komunikację, to dalsza wymiana informacji prowadzi do połączenia obu drzew. Jeśli jeden z tych węzłów nie jest korzeniem drzewa (nie prowadził on aktywnego poszukiwania innych węzłów, a jedynie pasywnie nasłuchiwał na określonym porcie UDP), to odsyła partnerowi numer węzła, będącego korzeniem jego drzewa i od tej pory komunikują się jedynie korzenie drzew. Komunikacja ma na celu ustalenie który z nich będzie korzeniem nowego, wspólnego drzewa: jeśli jeden z nich jest w danym momencie węzłem nadrzędnym replikacji drzewa, które cały czas świadczy usługi, to zostaje on korzeniem wspólnego drzewa; w innych przypadkach decyduje o tym stan repliki — korzeniem będzie ten, u kogo jest ona bardziej aktualna. Nowy korzeń znajduje w swoim drzewie miejsce do dołączenia nowego poddrzewa (starając się zrobić to tak, aby drzewo było możliwie jak najbardziej wyważone), po czym następuje właściwe połączenie drzew — nowa mapa całego drzewa jest rozsyłana w dół całego drzewa. Procesy zarządzające replikami w węzłach, w których nastąpiło właściwe połączenie, muszą oczywiście odpowiednio skonfigurować replikację w swoich serwerach MySQL.

4.4. Podsumowanie pracy zespołu (T.M.)

Niniejszy rozdział to kilka słów kierownika podsumowujących dokonania zespołu.

4.4.1. Osiągnięte wyniki

Niestety prace rozwojowe nie zakończyły się powstaniem działającej implementacji zaproponowanych rozwiązań. Pomysły dotyczące końcowego kształtu systemu bazodanowego zostały zawarte w niniejszej dokumentacji.

4.4.2. Aktywność członków zespołu

Zespół sześcioosobowy szybko okazał się być jedynie pięcoosobowym. Kolega Damian Adamik nie pojawił się nigdy na żadnym spotkaniu i nie brał żadnego udziału w pracach zespołu.

Pracę pozostałych członków zespołu należy ocenić jako pozytywną; zaangażowanie w projekt plasuje się gdzieś pośrodku skali, porównując je z innymi studenckimi projektami realizowanymi w kilkuosobowych grupach.

Każdemu z członków zespołu zdarzyło się spóźnić z jakimś elementem wspólnego projektu, lecz nigdy nie były to krytyczne opóźnienia. Sytuacja taka zdarzyła się jedynie na pierwszym terminie oddawania, gdy końcowy skład zespołu jeszcze się kształtował.

Dzięki dobrze wykonywanym kopiom zapasowym udało się bez uszczerbku dla pracy przeżyć awarię zespołowego repozytorium *Subversion*.